An Open Toolkit for Reverse Engineering Data Visualisation and Exploration

A. Telea [1], A. Maccari [2], C. Riva [2]
[1] Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
alext@win.tue.nl
[2] Software Technology Laboratory,
Nokia Research Center, Helsinki, Finland
alessandro.maccari@nokia.com, claudio.riva@nokia.com

Abstract

Maintenance and evolution of complex software systems (such as mobile telephones) involves activities such as reverse engineering and software visualisation. Reverse engineering provides a conceptual framework for describing the software understanding and concept abstraction processes. This framework is implemented up to different degrees by several reverse engineering tools. However, we found the architecture of most of these tools hard to adapt to the domain and problem specific requirements posed by our current practice in Nokia. We believe that the architecture of a reverse engineering tool should reflect the logical steps of the conceptual framework. We propose such an architecture and present a concrete reverse engineering toolkit that implements it. The toolkit provides a flexible way to build reverse engineering scenarios by subclassing and composition of a few basic software components. We pay special attention to the visual and interactive requirements of the reverse engineering process. We compare our toolkit with other existing reverse engineering visual tools and outline the differences. We show our plans to use it for further research on visualising the complex software structures that we extract from our products.

1 Introduction

Program understanding [9, 1, 21] is an essential part of maintenance and evolution of complex software systems. The term ”program understanding” means identifying the software artifacts that compose a certain system and the structure and semantics of their relationships. Reverse engineering is the part of program understanding that concerns with the extraction of the low level system implementation data and their presentation at the right abstraction level. Several tools support reverse engineering by providing several automatic and user-driven activities for data extraction and visual data presentation [6, 7, 15].

In practice, however, attempts to reverse engineer large systems usually reach some functional and/or structural limitations of these tools. Some tools focus on the domain modelling and the program analysis mechanics but provide little for the *examination* and/or user *editing* of the extracted information. Other tools provide extensive, sometimes exotic data visualisation and editing facilities [15, 7], but do not support program analysis operations or are hard to integrate with tools that perform this task.

To address the above problems, we propose a new architecture for RE tools. First, we identify the common support the various RE tasks demand from such a tool and propose a few ’agents’ a generic RE tool should support (Sec. 1.1). Our architecture implements these agents as loosely coupled object-oriented software components that can be subclassed and/or composed to customise the tool for a given domain-specific scenario (Sec. 1.3). Sections 2 and 3 present our architecture in detail. Section 5 shows and end-user view of our tool. Section 6 illustrates the use of our RE tool for the analysis of concrete software data from the industry. Section 7 concludes the paper with directions for future work.

1.1 Reverse Engineering Tasks

A past investigation detected five major tasks that a reverse engineering (RE) tool should support [1]. These tasks are, in increasing abstraction level order: program analysis, plan recognition, concept assignment, redocumentation, and architecture recovery (see also Fig. 1)). *Program analysis* is the basic task that any RE tool should support. RE tools should provide two main program analysis services: *construction* of the program layered representation by automatic and user-driven (interactive)

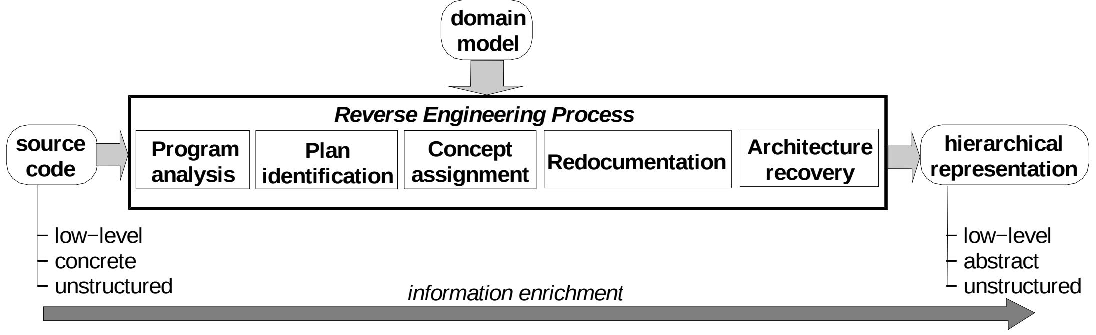


Figure 1: Reverse engineering tasks

operations, and the *presentation* of the layered representation to the user in various manners, such as graphical navigable views at different detail levels [15, 17]. *Plan recognition* aims at finding certain *design patterns* in the software [25]. These design patterns form the so-called *domain model* (Fig. 1). A first attempt for plan recognition support in a RE tool would be an interactive editor that allows the (manual) assignment of design patterns to elements obtained from program analysis and the visualisation thereof by means e.g. of UML diagrams. *Concept assignment* [8] is the task of discovering concepts and assigning them to their implementation counterparts in the subject system. RE tools might support concept assignment by annotating the software artifacts obtained from the previous stages with *concepts* picked from a domain-specific concept database and visualising this annotation. *Redocumentation* [21] is the task of retroactively providing documentation for existing software systems. Since redocumentation spans the three tasks discussed so far, a RE tool could support it by the mechanisms outlined so far. *Architecture recovery* [13] focuses on recovering the architectural aspects of large software systems. Architecture recovery support in RE tools may demand yet another (graphical) system representation as a set of subsystems.

1.2 Requirements for a RE Tool

Although different, the five mentioned RE tasks concur, and not compete, to the overall goal of reverse engineering, i.e. *extracting low-level information from the code* and *enriching it with information from other sources.* The unified requirements of these tasks could be summarised as: provide different views of the same data on which various queries, supported by a given domain model, can be made. Consequently, we identify three main 'agents' that a generic RE tool should provide: *views*, *queries*, and *domain models.*

The *views* relate to the different focuses of the RE tasks [1]. Among these, we mention *structural* views (syntax trees, file-directory containment hierarchies, design pattern instances), *functional* views (concept and dataflow graphs), *low-level* views (source code), and *high-level* views (architectural plans, user documentation). A generic RE tool should provide several of these views and allow defining and customising domain-specific views [14, 21].

The same flexibility should be provided for *queries.* A query is any algorithmic operation that can be executed on a given software representation, e.g. searching for occurrences of a specific pattern, removing or replacing of an element with a given value or structure, aggregation of several elements into a higher-level one, computing quality metrics, and so on. Since many queries are domain or application specific, RE tools should provide a generic and simple manner for defining them.

Finally, the RE tool should allow a simple way to define problem-specific *domain models.* A domain model is a set of meta-rules that describes certain aspects present in the analysed software, such as a source-code grammar, a set of design patterns to be identified, or a set of client-supplier relationships to be analysed. RE tools can use domain models to perform various queries automatically on the software data at a higher level than purely structural.

1.3 Concrete RE Operations

We approach the above requirements by designing a new reverse engineering toolkit. This toolkit addresses the needs for extensibility, domain retargeting, genericity, and simplicity to use by using a new architecture which closely models the steps of the conceptual RE pipeline, as follows.

We identify five generic operations that implement the five RE tasks introduced in Sec. 1.1, similarly to other authors [25, 14, 21] (see also Fig. 2):

1. *extract* the low-level artifacts from the source code
2. *aggregate* the extracted artifacts to produce a hier-

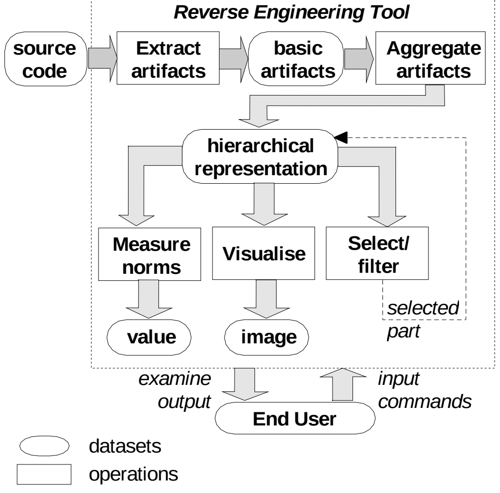


Figure 2: Reverse engineering pipeline

archical system representation

3. *measure* the quality of the produced representation by using various computed norms or by applying human intuition. If necessary, reexecute the aggregation step differently.

4. *select* a part of the hierarchy to examine, as the hierarchy's complexity and size precludes displaying it in full at one time. Moreover, selection (or filtering) allows users to focus on specific aspects of the representation.

5. *visualise* the selected data to gain a better understanding of the system. Insight acquired in this phase may determine to reexecute the previous phases differently.

Steps 2 to 5 can take place in any order - for example, one might want first to visualise the whole database produced by Step 1, then apply some user- or system-driven aggregation (Step 2), measure the quality of the obtained system (Step 3), select a certain feature to look at (Step 4), and so on.

Although traditionally being a program analysis model only, the above pipeline can be used all five RE tasks (Sec. 1.1). All selection, measuring, and visualisation operations apply thus to all five tasks, every task requiring potentially different operation implementations. Consequently, we use the above five-operation pipeline model as a basis for our generic RE tool's architecture (Fig. 3).

The toolkit is implemented as a layered system. Each layer is structured as a self-contained software component set. The toolkit core is implemented as a C++ class library, for performance reasons. The user interface layer is implemented in Tcl/Tk for flexibility reasons. The next

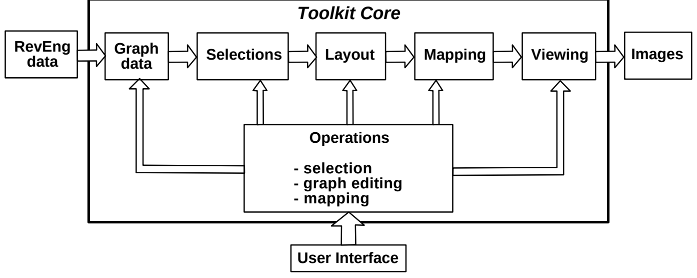


Figure 3: Overview of the toolkit architecture

section discusses the core data structures central to our architecture. The implementation of the RE operations is discussed next in Sec. 3.

2 Data Representation

We propose a data representation containing four elements: structure, attributes, selections, and domain models, as follows.

2.1 Structure

The structure data consists of two main elements:

- a representation of the *basic artifacts*, i.e. the low-level elements that constitute the input for the RE process, usually obtained by parsing the source code.

- a *hierarchical representation* of the structuring and enrichment of the information during the RE process.

There are many ways to describe this information. Korth [4] proposes a relational data model. Rohrich [12] discusses an attributed graph. The visualisation community discusses data tables [16, 17]. However, all these representations are ultimately *hierarchical attributed graphs*, best described as *(k,2)-partite graphs* [2] (Figure 4 shows an example. The software artifacts A,B,C,D, and E are read by the RE tool e.g. by parsing source code. This information is further refined by clustering A,B, and C into the F, D and E into G, leading to the hierarchy level 2 The nodes F and G are clustered into the node H, leading to the hierarchy level 3. All nodes A to H, linked by the horizontal edges (thick lines) and vertical edges (dotted lines), form the (k,2)-partite graph that completely describes the RE data structure.

(k,2)-partite graphs are particularly suited for our RE tool data model. They are *generic*, i.e. can store both the basic and the higher-level information produced by the RE data abstraction process. The graph levels may represent the *same type* of information, such as files in a file-directory hierarchy, or *different types*, such as variables,

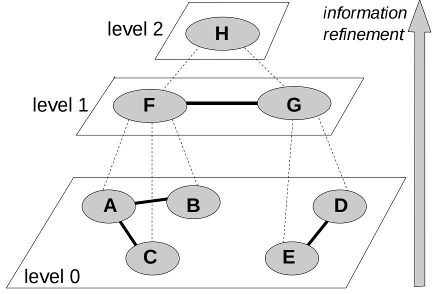


Figure 4: Example of (k,2)-partite graph

grouped in procedures, grouped in modules, etc. This representation covers all the five RE tasks: higher levels can represent architectural elements associated to syntactic software artifacts on the lower levels. The model is also easily *navigable* and *compact*, making it efficient for implementing the query, selection, filtering, and viewing operations. Graph nodes with several parents can easily model alternative system structurings.

2.2 Attributes

Both nodes and edges in the graph may have attributes modelled as key-value pairs. We implement keys as string literals and values as primitive types (integer, floating-point, pointer, or string). Each graph node and edge has an attribute-set, i.e. a collection of attributes with distinct keys. Managing attribute sets is optimised for simplicity: if one writes an attribute with a key that doesn't exist in the collection, a new attribute for that key is inserted, otherwise the existing attribute is overwritten. Attributes automatically change type if written to with a value of another type.

Several *attribute planes* can coexist in the graph. An attribute plane consists of all attributes of a given set of nodes and/or edges for a given key. For example, one can set or query the ”price” attribute-plane of a given node set.

The above model is quite different from the attribute representation of other RE and software visualisation tools [15, 6, 7]. Most such tools choose a fixed set of attributes of fixed types per node and/or edge, similarly to typed records. Our choice is more flexible, since a) certain attributes may not be defined for all nodes, and b) attribute-planes are frequently added and removed during a typical RE session.

Structure and attributes are implemented in terms of a `Graph` C++ class that has several `Level` objects (Fig. 5). A `Level` contains several `Nodes`, connected by horizontal `Edges`. `Nodes` and `Edges` inherit from the attribute-set class `AttrSet` that contains a number of `Attribute` instances.

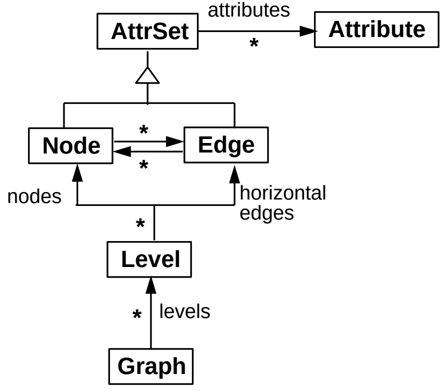


Figure 5: Graph dataset class hierarchy

2.3 Selections

Selections are the second main component of our RE data model. A selection is defined as a set of nodes and edges of the `Graph`. Selections allow executing any operation of our toolkit on a *specific subset of the whole graph*. To make the toolkit flexible, we decouple the definition of the operation from the selection on which it is executed.

The main property of selections is allowing their operation clients to iterate over the contained nodes and edges (Fig. 6). To optimise memory storage and iteration speed, we implement several selection subclasses that store nodes and edges internally in different ways. This is an important aspect, as an usual RE session creates tenths of selections with hundreds or even thousands of elements.

Selections are named, similarly to the attributes. All selections are kept in a selection-list, which is managed similarly to the attribute-set. Selections can be retrieved by name from the list. When a new selection is inserted into the list, any existing selection with that name is deleted. Asking for a selection whose name is not in the list returns a special `EmptySelection` object. In this way, operations do not have to test explicitly if their selection arguments exist or not in the list. This simplifies the implementation of a new operations.

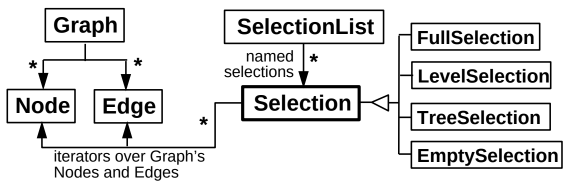


Figure 6: Implementation of selections

2.4 Domain Models

Domain models are implemented in our toolkit as a set of meta-rules that prescribe the structure and attributes of the nodes and edges. We implement a domain model as a collection of node and edge types. A node or edge type is a collection of attribute types. An attribute type consists of a name and a type. The above model is roughly similar to the type system provided by several programming languages for e.g. records. The toolkit supports a collection of domain models defined as above. Similarly to graph data, domain models can be loaded from several file formats, such as the RSF format [6]. Nodes and edges may then be associated with a node, respectively edge type by setting a ”type” attribute to name the desired attribute node, respectively. Operations such as metrics or aggregations may then use the types of nodes and edges to infer about their domain-specific properties.

Several RE tools such as [7] implement domain models differently, i.e. by modelling every node and edge type as a class in the toolkit's own programming language and implementing nodes and edges as instances thereof. However, this has several drawbacks. One can not add or delete attributes from existing nodes and edges, change attribute types, change the type of a node or edge, or introduce new domain models without recompiling the system. The main advantage of using a compiled solution — strong type checking — is not essential or may be even restrictive for the prototyping and investigative nature of the RE applications. We have chosen thus for a purely run-time, weakly-typed implementation of domain models.

3 Operation Model

Operations read and write the `Graph` dataset via `Selection` objects. Operations may have three types of inputs and outputs: *selections* that specify on which nodes and edges the operation works; *attribute keys* that specify on which attribute planes of the selected nodes and edges the operation works; and *parameters* specific to the operation, such as the threshold value for a data thresholding operation. Following the conceptual RE task model (Sec. 1.3), we distinguish three major operation types, as follows (see also Fig. 7). *Selection operations* encode different algorithms to build selection objects. Selection operations are the only ones that create selection objects and are described further in Sec. 3.1.

Graph	Selections	Operation
r	rw	selection
rw	r	editing
r	r	mapping

Figure 7: Operations versus reading/writing data

Graph editing operations modify the graph dataset by editing the graph's graph structure or graph attributes. These are the only operations that modify the `Graph` dataset and are described in Sec. 3.2. *Mapping operations* map the graph dataset to some other representations. Data visualisation operations and operations that save data e.g. to a file are mapping operations. These are the only operations that only read both the `Graph` and the selections and are discussed in Sec. 4.

The above read/write interface between operations and the system's data has two advantages. First, it provides a clear specification of the operations' responsibilities for the operation developers. Secondly, this allows the system to infer which data elements are modified after the execution of one or several operations and to automatically update all other system components that depend on the modified elements. For example, all selection objects are automatically updated after the execution of a structure editing operation, since this operation might have inserted or deleted nodes or edges referred by the selections. A second example are the data viewers presented in detail in Sec. 4.3 that are automatically updated once the selections they monitor get modified.

In the following, we describe several concrete operation implementations.

3.1 Selection Operations

Selection operations produce specific subclasses of the generic `Selection` interface (Fig. 6), as follows.

Level Selection
A level selection receives a level number as an input and produces a selection that contains all the nodes and horizontal edges on that level. Level selections can be useful, for example, as input for visualising a specific level of aggregation in a (k,2)-digraph that represents a software system. Level selections implement what other RE tools call *horizontal slices* in the data model [6, 7].

Tree Selection
A tree selection receives a selection s_1 as input and produces a selection s_2 that contains all the nodes and downward edges that are reachable from the nodes in s_1. Tree selections can be useful e.g. as input for visualising the so-called *vertical slices* in a (k,2)-digraph.

Conditional Selections
A conditional expression receives a selection s_1 as input and produces a selection s_2 that contains all the nodes and edges of s_1 that obey a certain condition. The condition can be specified by the user as a function of the node attributes. Conditional selections implement what the RE tools usually call *filtering.* They are used to e.g. visualise specific parts of a large graph in queries such as: 'show all nodes where the `cost` attribute is higher than a given

threshold'.

3.2 Graph Editing Operations

Graph editing operations edit the graph structure or the node/edge attributes, as follows.

3.3 Structure Editing

Structure editing operations construct and modify the `Graph` structure. The simplest operations are addition/removal of nodes, edges, and levels, implemented as methods of the corresponding classes `Node`, `Edge`, `Level`, and `Graph`.

3.3.1 Importing Graph Data

These operations read a graph structure from a file by parsing specific file formats such as RSF [6], GraphEd [20], and DOT [11]. Since the C++ code of these operations is 100 lines on average, it is reasonably simple to implement a reader for a new data format.

3.3.2 Aggregation

Aggregation operations usually take several nodes via an input `Selection` and produce a unique parent node. Implementing aggregation operations is straightforward. This involves iterating over the input nodes and constructing new vertical edges and a new node. The input selection can be either programmatically constructed or can be the output of user interaction (Sec. 4.2). More complex aggregation methods can be implemented too, such as topology-based automatic graph simplification [22].

3.4 Attribute Editing

These operations create, modify, and delete attributes from the nodes' and edges' attribute-sets (Sec. 2.2). Besides the `Selection` input common to most operations, attribute operations have also one or several attribute-plane names as inputs. These names refer to attribute-planes that the operation reads and/or writes, as follows.

3.4.1 Metrics

RE metrics are actually attribute editing operations. Examples of metrics include structural metrics, such as computing the number of provisions, requirements, and internalisations for a node [6, 1]. A metric can have two types of outputs: an *attribute-plane* for metrics that compute a new attribute for every node in the input `Selection`, or a *unique value* that characterises the input `Selection` as a whole, e.g. the cyclomatic number metric.

The above framework is significantly more flexible than the one provided by many existing RE tools. In short, it is easy to customise:

- the metric *implementation*. Usual node and edge based metrics have less than 10-50 lines of C++ code.
- the metric's *parameters*. The nodes and edges to apply the metric on are specified as selections created by any selection operation (Sec. 3.1). The attributes used to compute the metric on and to store the metric into are specified as attribute names. In this way, the metric implementation, the node and edge attribute implementation, and the selection implementation are all decoupled from each other.

3.4.2 Graph Layout

Graph layout operations are the basis of most RE data visualisations. These operations compute 2D or 3D coordinates for a selection of nodes and edges. Once such a layout is computed, the selected nodes and edges can be visualised, as described separately in Sec. 4.

We treat layout operations as attribute editing operations for several reasons. Computing a layout means, by definition, computing a position attribute for a set of nodes and/or edges. Treating the layout task as a normal attribute editing operations allows us immediately to:

- choose a desired layout type from a palette of available graph layouts.
- apply different layouts to different subsets of the graph. For example, a level-selection is best layed out with a spring embedder layout [18], whereas a call graph is best layed out with a Sugiyama layout [23, 11].
- combine different layouts. There is clearly no best layout available for any situation. Often it is best to construct a layout *incrementally*, i.e. apply several layouts sequentially to refine the same set of positions.
- modularise the toolkit implementation by decoupling the layout computation from the graph visualisation.

So far, we have implemented five layout operations, as follows (see also Fig. 8).

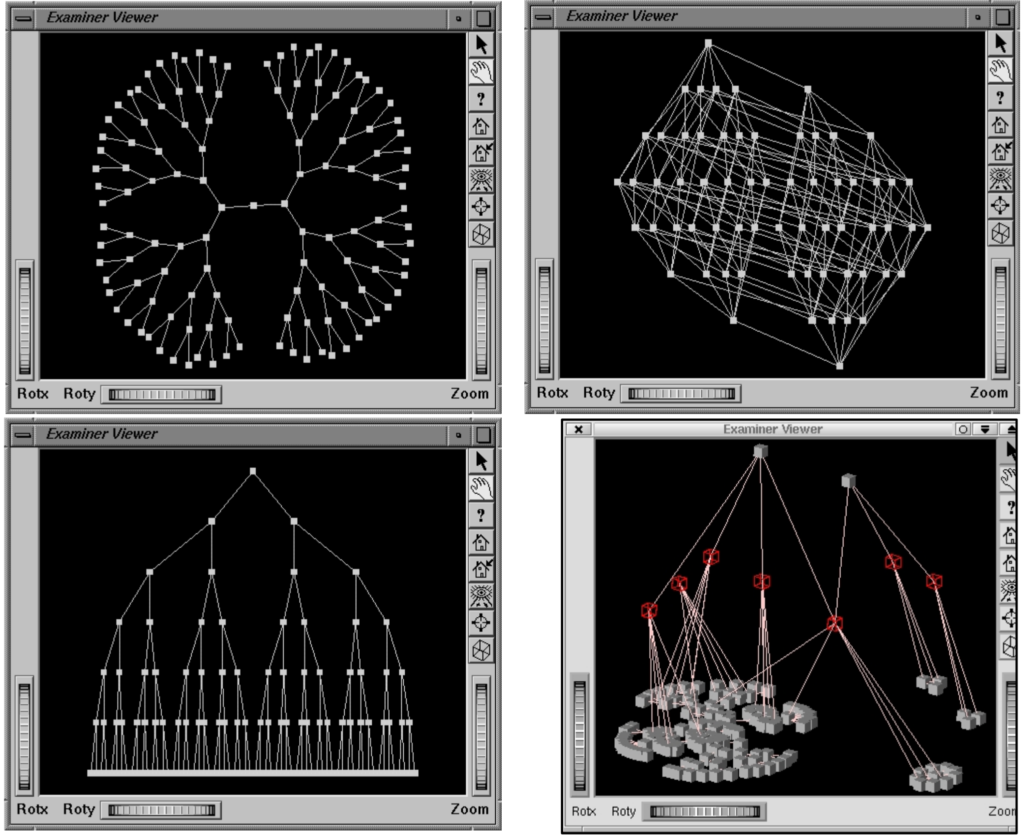


Figure 8: Software visualisation. Spring-embedder and tree-like layouts (top row). Vertical slice and 3D level layout (bottom row).

1. Tree-like layout: This layout treats the input selection as a directed graph. The selected nodes are arranged on levels in two dimensions such that the number of edge crossings is minimised [23]. We implement this layout based on the AT&T `dot` software [11]. We have conducted several tests out of which the `dot` software has run faster and more robustly for large graphs, and produced visually better layouts than other similar software such as [6, 7].

2. Spring embedder layout: This layout is based on a physical model that tries to minimise the edges' lengths with respect to the nodes' positions [10, 18]. We implement this layout based on the AT&T `neato` software provided by AT&T [10]. The performances of `neato` are similar to `dot`'s. We provide a second spring embedder layout based on the `GEM` algorithm [18]. Although functionally similar to `neato`, `GEM` implements different heuristics and has different control parameters than `neato`. We have sometimes experienced shorter running times and better visual results with `GEM`, especially for large, densely connected graphs. However, `GEM` is more sensitive to the layout parameters choice that `neato`. It is thus the toolkit user's choice whether to apply the one or the other.

3. Grid layout: This operation lays out the selected graph on a 2D regular grid. No attempt is done to minimise the edges' lengths or number of crossings. This layout is a fast, simple tool, optimal for small graphs or as a preprocessing phase to applying a more complex layout.

4. Random layout: This operation lays out the selected graph at random locations in 2D or 3D. The behaviour and usage of this layout is very similar to the grid layout.

5. Stacked layout: This operation lays out a selection spanning several levels of a (k,2)-digraph by applying one of the previous layouts per level and then stacking the layed out levels above each other in 3D. Stacked layouts are an effective way to visualise both the horizontal and vertical relationships of a (k,2)-digraph.

Adding new layouts to the toolkit is a reasonably simple task. The above layouts have been coded or encapsulated in no more than 100 C++ lines each, whereas the implementation of some layouts such as `dot` and `neato` exceeds 10000 C lines.

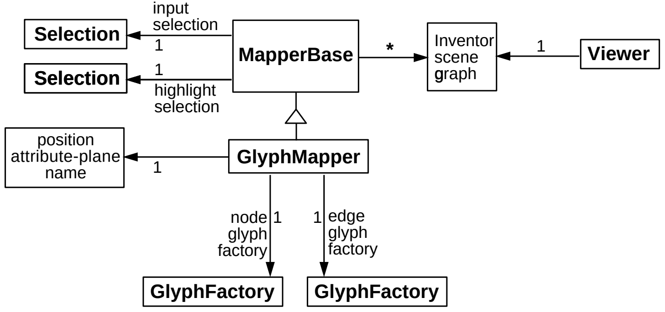


Figure 9: Components of the mapping operation

4 Mapping Operations

Mapping operations serve three purposes:

- *visualisation:* Mapping operations produce visual representations of the graph data. *interaction:* The visual representations can be interactively modified by the user.

- *export:* Mapping operations export the graph data to third parties e.g. for further processing or storage.

4.1 Data Visualisation

Visualisation operations have four components: mappers, viewers, glyph factories, and glyphs. These are implemented in our toolkit by the C++ classes with similar names (Fig. 9). Visualisation is based in our toolkit on the Open Inventor [28] graphics C++ library. As compared to other lower level graphics toolkits such as OpenGL, Inventor offers sophisticated mechanisms for object direct manipulation and programmatic construction that considerably simplify the programming of the visual interface of our toolkit. Figures 8 and 10 show several snapshots from graph visualisations created with our toolkit.

4.1.1 The Glyph Mapper

The central visualisation component is the mapper, implemented in our toolkit by the `MapperBase` C++ class. A mapper takes a `Selection` as input and produces an Inventor scene graph as output. The scene graph can be then interactively manipulated in several Inventor viewer windows. Specific `MapperBase` subclasses implement specific ways to construct the scene graph. One such subclass is the `GlyphMapper` that constructs a visual graph by creating one *glyph* for each node and edge in the input selection. These glyphs are then positioned at the 2D or 3D positions provided by an attribute plane of the nodes and edges. This attribute-plane is constructed prior to the mapping by a layout operation (see Sec. 3.4.2 and Fig. 3).

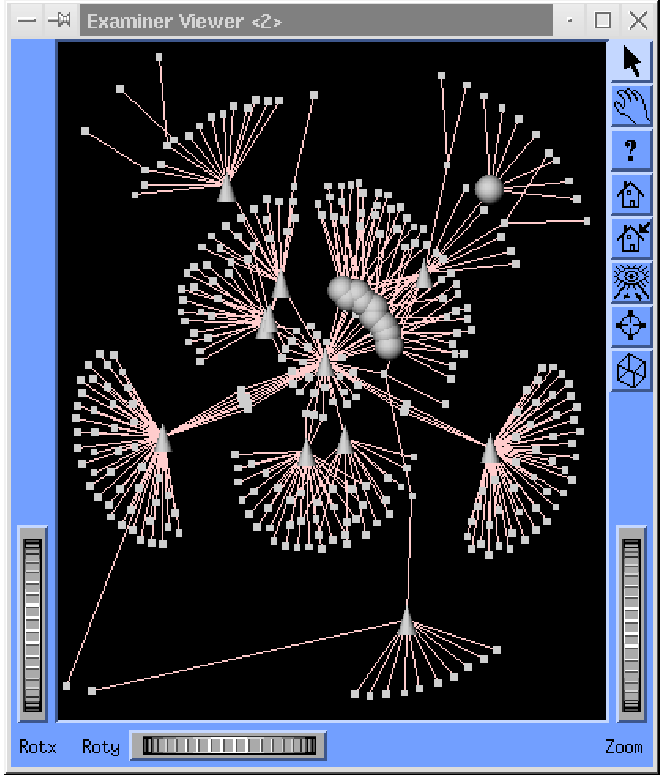


Figure 10: Usage of glyphs for software graph visualisation. Three types of glyphs (cones,cubes, and balls) are defined and associated with different node types. The glyph sizes reflect the nodes' number of edges.

4.1.2 The Glyphs

A *glyph* is a 2D or 3D graphical object that a `GlyphMapper` builds to visualise a node or edge. The glyphs' graphical properties reflect the node or edge attributes, as described next. This association is implemented by the `GlyphFactory` class. A `GlyphFactory` constructs a glyph given a concrete node or edge. Specific `GlyphFactory` subclasses build glyphs construction from nodes in various ways. For example, we provide a simple `DefaultGlyphFactory` class that builds a cube glyph for a node, respectively a line glyph for an edge. Every `GlyphFactory` subclass may provide a number of named parameters. The meaning of these parameters is specific to each subclass. For example, a `ConeGlyphFactory` subclass declares two parameters ”angle” and ”radius” that control the angle, respectively the radius of a 3D cone glyph. The user can associate these parameters with attribute-names to express which graph attributes are mapped to which glyph graphical parameters. Figure 10 shows a visualisation of a software system architecture in which the node type (package,class,function) maps to the glyph type (cube,cone,ball) and the node's number-of-lines attribute maps to the glyph size. Choosing the association between the node and edge attributes and their glyphs' graphical properties is a powerful manner to visualise the graph data [7].

Summarising, the `GlyphMapper` offers three ways to control the mapping of the graph data to visual objects:

- by specifying the `MapperBase` subclass that builds the Inventor scene graph; by choosing specific `GlyphFactory` objects that create specific visual representations for nodes and edges;
- by specifying the mapping between the graphical parameters of the `GlyphFactory` objects and the nodes' and edges' attribute names.

This design offers considerable freedom in producing a large class of visualisations. The separation of the glyph placement, done in the layout phase, and the glyph construction, done in the mapping phase, is a simple but powerful way in specifying the visualisation. New glyphs can be developed without any concern for how they are placed, whereas new layout tools can be added to operate on existing glyphs. Although relatively new in the software visualisation area, the mapper-glyph design has been used in the scientific visualisation community [26] where it has proven to be very flexible.

4.2 The Highlight Selection

Viewers function both as output components, by displaying their *input selection*, but also as input components, by editing a *highlight selection* (Fig. 9), as described next. The highlight selection is a subset of the input selection that is displayed in a special manner (the current implementation uses a special colour and drawing style). This is shown in the lower-right image Fig. 8 where the middle node layer has been selected by the user. The highlight selection can be edited interactively by the end users, as explained next.

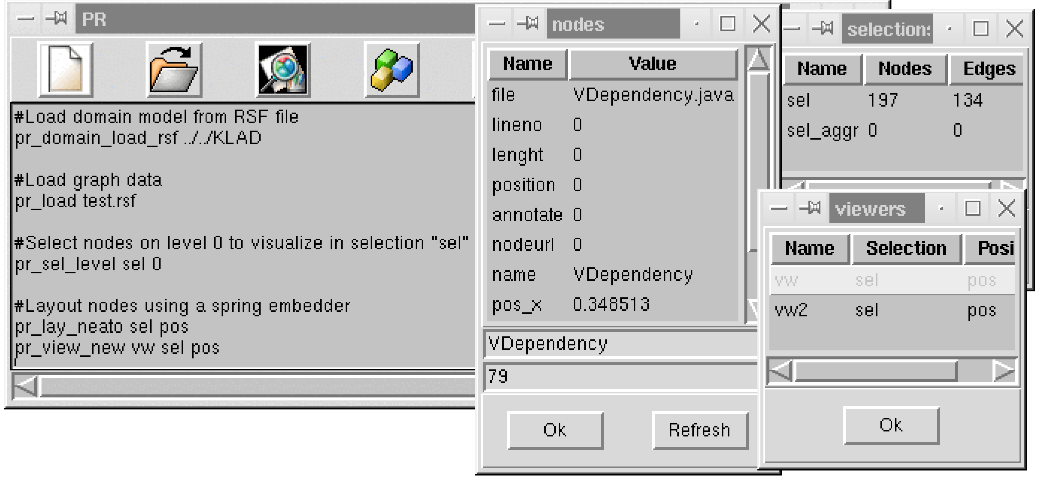
4.3 The Viewer

A viewer is an Inventor component that displays a scene graph constructed by a `MapperBase`. Several viewers are available that offer different mechanisms for interactive navigation in the visual space, the choice between parallel and perspective projection, and other visualisation customisations [28]. Viewers may have an input role too. Specific `Viewer` subclasses may respond to user events, such as mouse clicks an drags. `Viewers` use this mechanism to enable end users to edit, by mouse clicks, the highlight selection of their `MapperBase`s upon user interaction. As the highlight selection is automatically modified upon these events, other parts of the application can execute specific actions on the edited selection. For example, interactive node aggregation can be quickly implemented by applying an aggregation operation (Sec. 3.3.2) on the highlight selection object. Similarly, one can examine, delete, hide, interactively lay out, or apply metrics on the highlight selection.

5 User Interaction and Scripting

The toolkit core architecture described so far is implemented as a C++ class library. However, the main strengths of the toolkit are better leveraged when it is used within a dynamic prototyping environment. For this purpose, we provide a Tcl interface to the toolkit C++ API and add several custom Tk-based graphical user interfaces (GUIs) to build a complete integrated application. The GUIs provide simple access to commonly used functionality, such as examining and editing node attributes, selection objects, domain models, and viewers, loading and saving data, and so on (Fig. 11).

This GUI front-end is quite similar to RE integrated tools such as Rigi [6] or VANISH [7]. All these systems share an architecture based on a compiled core that provides a graph data representation and several operations, and a scripting and GUI layer built atop of this core. However, several differences are to be mentioned. First, our toolkit's core architecture is based on a few orthogonal components (graph, selections, operations, mappers, glyphs, and viewers) that have only simple interactions, and interdependencies (Fig. 3). Flexibility and custom adaptation are provided by subclassing and composition of these basic components. For example, the script shown in the background window of Fig. 11, used to produce the graph visualisation shown in Fig. 10, contains about 10 Tcl lines. In contrast, Rigi [6] uses a monolithic core architecture. Although adaptable via Tcl scripts, this architecture offers no subclassing or composition mechanisms *for the core itself*. It is not possible, for example, to change the graphic glyphs or the interactive selection policy without recoding the core. Similarly, adding a new layout algorithm, domain model representation, or metric involves a low level API to access nodes and edges, as Rigi has no notion of manipulating these as selections. In contrast, VANISH [7] provides a way to build custom glyphs very similar to our `GlyphFactory` class (Sec. 4.1.2). However, VANISH uses domain models based on compiled C++ classes which prove inflexible for our targeted RE scenarios (Sec. 2.4).



PR

```
#Load domain model from RSF file
pr_domain_load_rsf ../../KLAD

#Load graph data
pr_load test.rsf

#Select nodes on level 0 to visualize in selection "sel"
pr_sel_level sel 0

#Layout nodes using a spring embedder
pr_lay_neato sel pos
pr_view_new vw sel pos
```

nodes

Name	Value
file	VDependency.java
lineno	0
lenght	0
position	0
annotate	0
nodeurl	0
name	VDependency
pos_x	0.348513

VDependency

79

Ok Refresh

selection:

Name	Nodes	Edges
sel	197	134
sel_aggr	0	0

viewers

Name	Selection	Posi
vw	sel	pos
vw2	sel	pos

Ok

Figure 11: Tcl/Tk interface of the integrated reverse engineering application

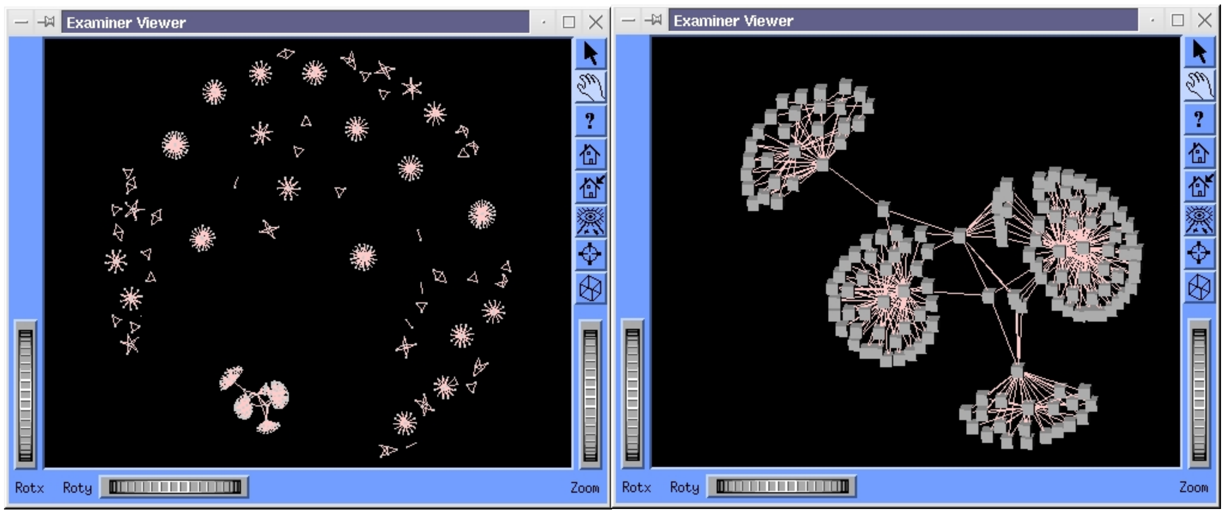


Figure 12: Mobile telephone software visualisation. Entire system (left) and detail (right)

6 Applications

We have used the presented integrated GUI application for the exploration of reverse engineering data obtained from the software systems built at Nokia. In the first pass, an attributed graph is extracted from the original Java program source code. This graph contains various low-level software entities such as functions, classes, files, and packages, which represent the graph nodes. Relationships such as ’uses’, ’contains’, and ’calls’ are encoded by graph arcs. Finally, various attributes such as object names, number of code lines, version numbers, etc are stored as node attributes. Next, this graph data is loaded in out RE tool as a RSF file [6]. From this point, the operation pipeline described in Sec. 1.3, i.e. selection, aggregation, metric computation, layout, mapping, and viewing, is executed.

Figure 12 (left) shows a selection of 10% of the whole extracted graph, visualised with a spring embedder layout. It is easy to see that the about 900 software entities shown in this view are grouped in several independent clusters that correspond to the different subsystems in the original software. The largest subsystem, shown in the lower left part of the figure, was selected interactively by the user and then displayed separately in a second viewer, shown in the right image. In this image, we can easily detect the ’bridge’ components of the software system as being those nodes that connect the large, densely coupled subgraphs.

The whole visualisation scenario, starting from the RSF data delivered by the code analyser, took about 5 minutes to build. This implied the execution, via both the tool’s GUI and its Tcl command-line input, of less than 20 operations of the ones described in Sec. 1.3. To produce the zoomed-in image shown on the right, a Tcl procedure of less than 15 lines was written that takes the highlight selection output of the left viewer (Sec. 4.2, applies a spring embedder layout, maps it, and views it in a new viewer instance. To visualise node details, such as attribute names and values, we have written a second 12-line Tcl procedure that opens a inspection GUI window (as shown in Fig. 11 middle). this procedure is activated on the changing of the highlight selection in the detail viewer. Visualising node details is thus easily done by clicking on the desired node in this viewer.

Overall, the usage of our RE tool proved to be more flexible than using the Rigi system, for the same case data, both for the end user that issues GUI or Tcl commands and for the developer that adds new functionality to the tool. The most time-consuming part of the executed scenarios was the layout computation. The layout implementations we use work reasonably fast for a few hundreds of nodes. For larger graphs, one has to apply several selection and/or aggregation operations to reduce the size of the data, prior to the layout and visualisation stages.

7 Conclusion and Future Work

we have presented a new architecture for building visual RE tools. The presented architecture makes it easy to construct several RE scenarios by compositing and/or subclassing a set of given software components. These components model the data and operations present in the abstract RE framework described by several authors. RE data is represented as a generic attributed graph, which allows a flexible way to define most data gathered from the program analysis stage. The presented architecture classifies the RE operations in graph editing, selections, layout, glyph mapping, and viewing. Functionally, these operations model respectively structural aggregation and metrics computations, queries and filtering, and constructing a visual representation for the data.

The flexibility of the presented architecture reflects itself on two levels. First, end users can easily define custom data investigation scenarios simply by applying the provided operations in different orders with different parameters. Our architecture supports this scenario, as operations are only indirectly coupled by sharing the same graph data structure. Secondly, developers can easily define custom operations by subclassing the already implemented operations or operation interfaces. The loose coupling between operations makes their code localised and limited in complexity - the about 40 operations we have implemented have each on the average 20 to 40 lines of C++ or Tcl code. The architecture promotes a separation of concerns for the different operation types that makes it natural for developers to write small, independent operations. Usually written in C++ for performance, such core operations are then easily assembled into larger, more specific operations written in interpreted Tcl.

Overall, the implementation of the presented architecture has about 8000 C++ lines grouped in around 50 classes and took four man-months development time. The toolkit implements around 30 operations (5 data readers, 4 data writers, 12 structure editing and metrics operations, 6 layout operations, and about 10 mapping operations). The GUI-based integrated application built atop of the toolkit adds around 500 Tcl lines to the C++ core and is already easier to customise and use than other systems with a more rigid architecture, such as [6, 7]. As a last comment, we believe that the analysis of the architectural aspects involved in building RE tools performed in this paper is important for the development of flexible, customisable RE applications.

Our main future work is targeted at providing domain-specific operations, such as graph simplification, layout, and glyph mapping, for the domain models used at Nokia for describing their current software architectures. Our RE system will thus serve both as a tool for investigation of the concrete mobile telephony software and as a testbed for prototyping new information visualisation techniques.

References

[1] S. TILLEY, *A Reverse-Engineering Environment Framework*, Technical Report CMU/SEI-98-TR-005, Carnegie-Mellon University, 1998.

[2] H.A. MULLER, M.A. ORGUN, S. TILLEY, J. UHL *A Reverse Engieering Approach to Subsystem Structure Identification*, Software Maintenance - Research and Practice, 5(4), pp. 181-204, 1993.

[3] J. EDACHERY, A. SEN, F. BRANDENBURG, *Graph Clustering using Distance-k Cliques*, Proc. Graph Drawing '99, LNCS 1731, pp.98-106, 1999.

[4] H. KORTH AND A. SILBERSCHATZ, *Database System Concepts*, McGraw-Hill, 1986.

[5] T. ROXBOROUGH AND A. ARUNABHA, *Graph Clustring using Multiway Ratio Cut*, in Proc. Graph Drawing 1996, Springer-Verlag, 1996.

[6] K. WONG, *Rigi User's Manual version 5.4.4*, Dept. of Computer Science, University of Victoria, Canada.

[7] R. KAZMAN, J. CARRIERE, *Rapid Prototyping of Information Visualizations using VANISH*, Proc. IEEE InfoVis '95, IEEE CS Press, 1995.

[8] T. BIGGERSTAFF, B. MITTBRANDER, D WEBSTER, *The Concept Assignment Problem in Program Understanding*, Proc. Working Conference on Reverse Engineering WCRE '93, IEEE CS Press, 1993.

[9] P. YOUNG *Program Comprehension*, Visualisation Research Group, Centre for Software Maintenance, University of Durham, May 1996.

[10] S. C. NORTH, NEATO *User's Guide*, AT&T Bell Labs Report Series, `http://www.research.att.com`, 1996.

[11] E. KOUTSOFIOS, *Drawing graphs with* DOT, AT&T Bell Labs Report Series, `http://www.research.att.com`, 1996.

[12] J. ROHRICH, *Graph Attribution with Multiple Attribute Grammars*, ACM SIGPLAN 22 (11), pp.55-70, 1987.

[13] K. WONG, S. TILLEY, H. MULLER, M. STOREY, *Structural Redocumentation: A Case Study*, IEEE Software 12 (1), 1995, pp. 46-50.

[14] K. WONG, *On Inserting Program Understanding Technology into the Software Change Process*,internal report, Dept. of Computer Science, University of Victoria, Canada, 1997.

[15] S. EICK AND G. WILLS, *Navigating large Networks with Hierarchies*, in *Readings in Information Visualization* [16].

[16] S. CARD, J. MACKINLAY, B. SHNEIDERMAN, *Readings in Information Visualization - Using Vision to Think*, Morgan Kaufmann Publishers Inc., 1999.

[17] J. STASKO, J. DOMINGUE, M. H. BROWN, B. A. PRICE, *Software Visualization - Programming as a Multimedia Experience*, MIT Press, 1998.

[18] A. FRICK, A. LUDWIG AND H. MEHLDAU, *A fast adaptive layout algorithm for undirected graphs*, Proc. Graph Drawing'94, Springer-Verlag 1995.

[19] E.R. GANSNER, S.C. NORTH, *An open graph visualization system and its applications to software engineering*, Software-Practice and Experience, John Wiley & Sons, (S1) 1-5, 1999.

[20] M. HIMSOLT, *GraphEd user manual*, Technical report, Fakultat fur Informatik, Universitat Passau, Innstrasse 33, D-8390 Passau, 1992.

[21] S.R. TILLEY, K. WONG, M.D. STOREY, H.A. MULLER, *Programmable reverse engineering*, Dept. of Computer Science, University of Victoria, Canada, 1998.

[22] J. TOLLE, O. NIGGEMANN, *Suporting Intrusion Detection by Graph Clustering and Graph Drawing*, Proc. Graph Drawing '98, Springer-Verlag, 1998.

[23] K. SUGIYAMA, S. TAGAWA, M. TODA, *Methods for Visual Understanding of Hierarchical Systems Structure*, IEEE Trans. Systems, Man, and Cybernetics, Vol. 11, No. 2, pp.109-125, 1989.

[24] M.D. STOREY, H. A. MULLER, *Graph Layout Adjustment Strategies*, Dept. of Computer Science, University of Victoria, Canada, 1998.

[25] A. MENDELZON, J. SAMETINGER, *Reverse Engineering by Visualizing and Querying*, internal report, Computer Systems Research Institute, University of Toronto, Canada, 1997.

[26] W. SCHROEDER, K. MARTIN, B. LORENSEN, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, 2nd edition, Prentice Hall, 1998

[27] S. EICK, J. STEFFEN, E. SUMNER, *Seesoft-tool for visualizing line oriented software*, IEEE Trans. Software Engineering, 11(18),pp. 957-968,1992.

[28] J. WERNECKE, *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Addison-Wesley, 1993.